THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 60, No. 7, September 1981
Printed in U.S.A.

Digital Signal Processor:

Design of the Assembler

By C. L. SEMMELMAN

(Manuscript received June 13, 1980)

In addition to the features normally provided by assemblers, the digital signal processor assembler handles the multistatement-per-instruction format required by a pipelined machine and provides several other capabilities required by the digital signal processor architecture. In describing the manner in which this was accomplished, more attention is devoted to matters of interest to the user of the assembler and less to its internal construction. The use of "lex" to write the parser subroutine is described, and possible future enhancements are discussed. An example illustrates the digital signal processor assembler's input language and its outputs.

I. INTRODUCTION

The assembler for the digital signal processor (DSP), as for any processor, converts programs written in a symbolic language into the corresponding machine language, and provides various convenience features for use by the DSP programmer.

The architecture of the DSP is designed for a maximum speed of operation in applications which differ markedly from those of ordinary computers. As a result, the DSP assembler contains features which are unique to this DSP application. At the assembly language level, this results in a complex programming language, which the programmer must understand thoroughly in order to produce correct and efficient programs. The assembler, of course, must accept every legal instruction in this input language and produce machine language corresponding to every operation in the DSP repertoire. This language differs from standard assembly languages in several major respects, as described below.

II. ARCHITECTURAL FEATURES OF THE DSP AND THEIR EFFECT ON THE ASSEMBLER DESIGN

The DSP has a number of unusual architectural features which affect the design of the assembler. Following an overview, these features are described from a user's point of view and their effects on the assembler's design and operation are discussed. Boddie et al.[1] give a more complete description of the machine architecture.

2.1 Overview of DSP architecture

The DSP contains 1024 words of 16-bit ROM memory for program storage and 128 words of 20-bit RAM memory for data. The processor contains a 16- by 20-bit integer multiplier, whose output is fed into a 40-bit integer accumulator. From there, data may be sent to a 20-bit W register, before being sent to storage or back to the multiplier for further calculation. Input and output are handled through 8-bit buffers, with automatic serial-parallel conversion and external synchronization. The DSP has four kinds of special purpose registers, including five registers for indirect addressing of data (direct addressing is not allowed), four registers for increments and counting, two for setting the DSP ground rules, and others for instruction counting, return address storage, synchronization, and status output. A separate adder is used to increment the addresses stored in the indirect addressing registers.

The multiplier, accumulator, W register, and data storage functions are separately programmed and controlled by different fields in the machine language word. There are two different classes of instructions: arithmetic and auxiliary, and they make different uses of some of the bits of the machine language word. The DSP uses the value of one of the instruction fields to distinguish between the two classes. Arithmetical instructions occupy only one machine word and may specify that the next machine word contains a numerical value for immediate input to the multiplier. Auxiliary instructions occupy two machine words, and bits in the second word further distinguish between arithmetical-auxiliary and non-arithmetical-auxiliary subclasses.

2.2 Pipeline architecture

The term "pipeline" refers to the fact that the processor has several hardware components which perform different operations simultaneously and pass data from one component to the next as through a pipe. In the DSP, these components are a multiplier, an accumulator, the W register and a memory. Data flow from the multiplier to the accumulator, then to the W register and to storage. Each of the hardware

components is under the command of a specific group of bits in the instruction, which selects the exact operation to be performed out of the group available to the particular component.

Figure 1 shows how the pipeline functions. The columns correspond to the hardware components, and the rows to instructions which are executed sequentially in time. Data move diagonally. Each component can accept and process data in each instruction cycle, if commanded to do so. Although it takes four cycles for data to progress through all four components, the DSP accepts a new input argument and produces a new output each instruction cycle. This increases the data processing speed appreciably.

For each instruction, the assembler must accept up to four statements, one for each hardware component, and combine the corresponding bit patterns to form the complete instruction. The statements are usually written in the left-to-right sequence shown in Fig. 1, as this encourages the following interpretation:

(*i*) Store the present contents of the W (or Y) register.

(*ii*) Reload the W register from the accumulator.

(*iii*) Reload the accumulator using the present contents of the product register.

(*iv*) Calculate a new product from the x and y arguments specified.

An instruction containing four such statements might be

```
*rda = w   w = a   a = p + a   p = *rx++j * ibufy.
```

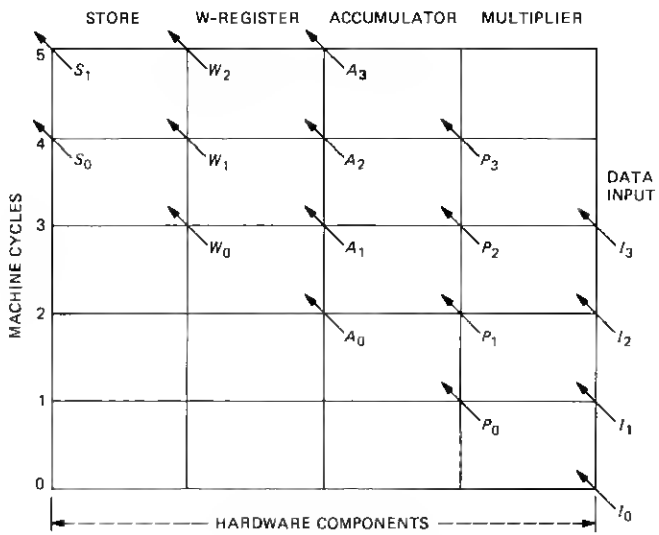

Fig. 1—Pipelining of data.

2.3 Advanced fetches of x and y fields

The x and y fields in a DSP instruction specify the two input arguments to the multiplier. The sources for the x and y data must be specified in the instruction that is fetched two cycles before the instruction that operates on the data. The assembler accepts the x and y source specifications on the same line as the operation (as shown in the previous example) and then "skews" them to the x and y fields of the instruction two instructions previous. The four-statement instruction shown above would appear in machine language as in Fig. 2.

2.4 Instruction fetched two cycles before execution

Each instruction is fetched from read only memory (ROM) two cycles before it is executed. This allows time to decode the remaining instruction fields before execution begins. If this only resulted in a two-cycle time delay between fetching an instruction and executing it, programmers would be able to ignore the delay completely. Unfortunately, however, this delay in execution also applies to jump instructions. The two instructions that follow a jump are already in the operating hardware when the jump takes effect and their x and y fields will affect instructions that follow the jump. They may differ from the x and y fields that would be fetched if the jump destination were reached by normal program counter incrementing. Fields $x1$ and $y1$ and fields $x3$ and $y3$ in Fig. 3 both refer to the same operation instruction located at the destination.

The current version of the assembler cannot determine if these two sets of x and y fields should be alike or if they may be different. The programmer must answer this question. The assembler tests and reports a difference as a warning message.

Both the advanced fetching of the x and y fields described above and this instruction fetching in advance of execution are forms of pipelining, and they create problems quite unlike those encountered in the writing of standard assemblers.

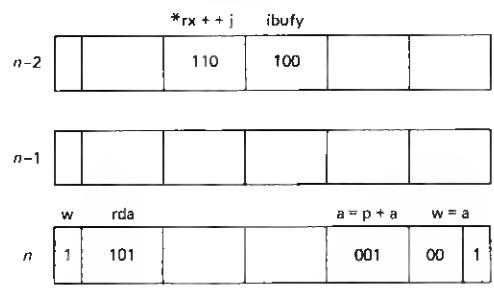

Fig. 2—Fields x and y advanced by two instructions.

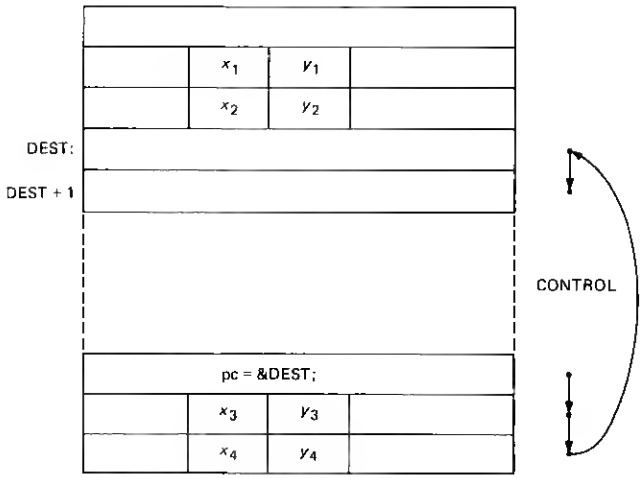

Fig. 3—Multiple x and y fields apply to same instructions.

2.5 Other hardware features

The other hardware features mentioned above cause few problems for the assembler and even eliminate some of the usual assembler functions. Input and output are treated as if they were two storage locations. The special purpose registers appear in the source program only to be loaded with a constant. When they are used for other purposes a key word, which also specifies the application, conceals their identifiers. Even statements specifying arithmetical operations do not have to be analyzed as the assembler recognizes the entire statement and translates it into a single bit pattern. For example, the entire statement

```
a = p + a
```

is identified as one key "word" and translated into the binary number 001. A multiplication statement, such as

```
p = *rx++j * ibufy,
```

contains two key words, *rx++j and ibufy, which become 110 and 100. Multiplication is implied and there is no place for the product to go except into the product register, so it need not be specified.

2.6 Summary

The unusual architecture of the DSP produces some strange results both in the assembler operation and in the appearance of source programs. The frequent use of key words and the prohibition of direct addressing combine to make variable names almost disappear from the source program. Their only use is to prime registers for indirect

addressing. Pipelining also helps to conceal the programmer's intent. As a result, source programs are more difficult to read and good comments are more important than in most assembly language programs.

III. EXTERNAL FEATURES OF THE DSP ASSEMBLER

In the development of the assembler, a number of policy questions were considered and settled before program writing was started. These decisions are discussed below.

3.1 Environment for development and customer use

Because of the widespread use of the *UNIX** time-sharing operating system[2] at Bell Laboratories, no alternative was given serious consideration. This choice makes it easy for programmers to prepare their DSP source language programs using the *UNIX* text editor, store them as files, and have the assembler pick them up for processing. Assembler output files can be listed and retained for testing with the DSP simulator,[3] and can be converted to PROM or ROM mask formats.

3.2 Special DSP hardware features

The programmer is required to place the individual statements which make up an instruction, so that the pipeline operations will be performed on the correct data. The programmer must also place jump statements properly, as the assembler does not advance them by two instructions. This is considered proper as the programmer must understand the effects of the advanced x and y field fetches.

The assembler does advance the x and y fields for the programmer. This results in a more readable source program and eliminates one source of possible error in the assembler input. The assembler also deduces, from the key words found, which of the various instruction classes and sub-classes the programmer is using.

3.3 Input language characteristics

The input language syntax resembles that of the language C. This choice was made because many DSP programmers were familiar with that language.

To facilitate programming for people who do not need the full mnemonic content of the C-like constructs, a set of three-character alternates is available. Thus, a programmer has the choice of entering either

```
*rx++i      or      rxi
```

* Registered trademark of Bell Laboratories.

to mean "the quantity pointed to by the contents of the RX register, which is post-incremented by the contents of register I."

The assembler also provides a limited macro facility. It permits a single instruction macro to be defined and called within the source language input file. Argument substitution is not implemented. For more elaborate facilities, the programmer may use the C-language preprocessor, which allows multi-instruction capability, nesting of macros, and argument substitution.

3.4 Handling of errors

The assembler reports each error in a message which appears on the user's terminal. Each error is classified as "fatal" or "for information," and, where possible, each message contains the number of the line in the input file where the error was detected. Any fatal error will prevent the writing of the PROM programming file, and no error or combination of errors is capable of stopping the assembler before it reaches the end of the source file.

IV. IMPLEMENTATION POLICIES AND DETAILS

The programming methods described here were selected because they were convenient in developing the assembler and led to good quality code.

4.1 One-pass assembler

Because so few variable names are expected in the DSP programs, it seemed reasonable to assemble as much of the program as possible on the first pass over the source file. This decision forces the programmer to define macros before calling them and to assign variables to RAM storage before referring to them, which appear to be reasonable restrictions.

The assembler handles labels in the following manner. When a label definition is encountered, the assembler puts the memory address and the current ROM address in a label definition table. When a label reference is encountered, the memory address and the ROM location of the reference are added to a table of label references. At the end of the source program file, the assembler moves the label definition ROM locations into the corresponding label reference locations. Any undefined references cause fatal errors.

4.2 Organization of the DSP assembler

A very brief description of the assembler is as follows: the main program calls the parser, which is a subroutine written by "lex."[4,5] It returns a value identifying the token found. Control passes through two levels of "switch" statements to a block of code where the bits

corresponding to the token are moved into the machine word, and some flags are set or some table entries made. These actions are repeated until the end of file is encountered. Then label references are resolved, output files written, and messages written for the user.

The following paragraphs elaborate on this brief description, but still give only an overview of the methods used. Tables I through VIII in the Appendix show the statements which may appear in each class of instruction and the tokens which are permitted in each statement. Reference to these tables may be helpful in reading the following paragraphs.

4.2.1 Token identifiers

The numerical values used to identify the tokens are octal numbers whose "hundreds" digit identifies a family of tokens and whose "tens" and "units" digits indicate the member of that family. In addition to the families of tokens shown in Appendix A, there is a utility family, whose members are semicolons, label definitions, macro starts, ends and calls, numerical values, comments, RAM assignments, dimensions, and subscripts. Some of these items are described more fully in the next section.

4.2.2 Utility functions

Several members of the utility family are described below. Label definitions have already been discussed, and comments, dimensions, and subscripts do not need extensive coverage.

4.2.2.1 Semicolon. The semicolon is used to mark the end of each DSP instruction. Its appearance initiates the clean-up after one instruction and the priming required for the next.

4.2.2.2 Macros: start, end, and call. A macro is defined as in the following example:

```
{Macname rdx = y   a = p   p = axi * ryk},
```

where the braces signal the start and end of the definition. At the left brace, the parser is called to read the macro's name, which is saved in a table. The statements in the macro are assembled in a macro table location, rather than in their final ROM position. The right brace causes the assembly location to revert to the normal ROM position.

Mentioning the name of the macro causes the saved bits to be placed in the proper position. The DSP programmer can add additional statements to the macro before the semicolon completes action on that instruction.

4.2.2.3 RAM variables. A RAM assignment statement appears as follows:

```
ram Z, ABC, TABLE[10];
```

The key word "RAM" causes control to go to a block of code where

each token is read. The assembler adds each variable name, its dimension which defaults to 1, and the RAM location assigned to it, to a table of RAM variables. At the semicolon, control returns to the normal loop.

4.2.2.4 Numbers. Numbers may be used for several different purposes: as an immediate value to act as the x input to a multiplication, to build a table in ROM memory, to load a register, as a dimension in a RAM variable definition or as a subscript in a RAM variable reference. Thus, the processing of a number depends on the context of its use.

4.2.3 Flags and error detection methods

Fifteen different flags are used in the assembler. Among their uses are recording the presence of a member from each family of tokens, the arithmetical or auxiliary class of an instruction, and whether the instruction occupies two ROM words or one. The flags are used in tests for correctness of the source code and in steering the assembler.

The tests for errors are quite thorough. They may include some tests for errors which can never occur, as it was easier to include the test than to prove the impossibility of the error. The parser rules include one which detects any character which is not a letter, a number or one of the specified group of punctuation symbols. When such a character appears, a message to the user is sent and a fatal error recorded.

4.2.4 The file 'dsp.listing'

This file is the output medium by which the assembler communicates its results to the DSP programmer. A sample of the output is shown in Fig. 6. The tables referring to macros and labels are prepared initially as separate disk files and later concatenated with the file containing the remainder. This file is prepared by rewinding and rereading the input source program, matching each line to the assembled code.

V. RESULTS

An assembler for DSP programs has been written and functions as described in the preceding paragraphs. The assembler contains about

```
# include "dspbq.h"
"Two Biquad Sections in Cascade"
"Dial Tone Rejection Filter"
ioc = 0502;        /* 8 bits in, 16 out */
auc = 067;
i = 1;
j = -1;
k = -3;
loop: syc = 1;
bqic
bquad (1.,-1.957,1., -1.112,.544)
bquad (1.,-1.891,1.,-1.7328,.94297)
bqnoc (loop)
```

Fig. 4—Input file for biquadratic filter using macro cells and C-preprocessor.

```
Completed scan of source file. 0 fatal errors.
Files b.out and d.out were written to disk
dsp.listing was written to disk.
To save files, mv them to other names.
```

Fig. 5—Digital signal processor assembler messages to user in a successful assembly.

3500 lines of code and comments and has been in use for over a year in a wide variety of applications. Its output has been used as input to the simulator program,[5] thus, assuring compatibility.

VI. FUTURE ENHANCEMENTS

Enhancements to the DSP assembler fall into near-future and more remote-time categories.

```
D.S.P. ASSEMBLER, [data of version]
    Two Biquad Sections in Cascade
    [data and time of assembly]

    MACRO DEFINITIONS
    MACRO NAME   ADVCMD   CMD    DATA   W/A   SOURCE
                  octal    octal   octal

    LABELS
    ROM LOC.   LABEL      REF. AT LOC.
      dec.                  dec.
       10      loop           44

    RAM VARIABLES
    RAM LOC.   DIMENSION   NAME
      dec.       dec.

LOC.  COMMAND    DATA     X   Y   LINE                    SOURCE
dec.   octal      octal            dec.

                                          "Two Biquad Sections in Cascade"
                                          "Dial Tone Rejection Filter"
 0 : 00 00 10 : 15 0502 (aux w  )   3 AN  ioc = 0502;
 2 : 00 00 10 : 14 0067 (aux w  )   4 AN  auc = 067;
 4 : 00 00 10 : 10 0001 (aux w  )   5 AN  i = 1;
 6 : 00 00 10 : 11 7777 (aux w  )   6 AN  j = -1;
 8 : 00 00 10 : 12 7775 (aux w  )   7 AN  k = -3;
10 : 00 00 10 : 16 0001 (aux w  )   8 AN  loop:  syc = 1;
12 : 00 04 10 : 02 0000 (aux iny)   9 AN           ry = 0;
14 : 00 00 10 : 04 0000 (aux w  )  10 AN           rd = 0;
16 : 00 21 01 : 000020 (imm ryi)   11 AA                                  p = mtl1(iny);
18 : 00 22 01 : 000031 (imm ryj)   12 AA                        a = p     p = mtl2( );
                                          obuf = w              a = p + a
20 : 14 21 10 : 156457 (imm ryi)   14 N            p = -.544* ryi;
22 : 00 21 10 : 043453 (imm ryi)   15 N                        a = p + a  p = -1.112*ryj;
24 : 00 20 10 : 040000 (imm w  )   16 N                        a = p + a  p = 1.*ryi;
26 : 01 21 01 : 101301 (imm ryi)   17 N  rdi = y   w = a       a = p      p = -1.957*ryi;
28 : 11 22 10 : 040000 (imm ryj)   18 N  rdi = w               a = p + a  p = 1.*w;
                                                               a = p + a
30 : 00 21 10 : 141646 (imm ryi)   20 N            p = -.94297* ryi;
32 : 00 21 10 : 067346 (imm ryi)   21 N                        a = p + a  p = -1.7328*ryj;
34 : 00 20 10 : 040000 (imm w  )   22 N                        a = p + a  p = 1.*ryi;
36 : 01 00 01 : 103372 (aux w  )   23 N  rdi = y   w = a       a = p      p = -1.891*ryi;
38 : 11 00 10 : 040000 (aux w  )   24 N  rdi = w               a = p + a  p = 1.*w;
                                                               a = p + a
40 : 00 00 11 : 000001 (aux w  )   26 AA           ;
42 : 00 00 01 : 000100 (aux w  )   27 AA           w = a;
44 : 00 00 10 : 00 0012 (aux w  )  28 AN           pc = &loop;
46 : 00 00 00 : 000000 (aux w  )   29 AN  ; ;
48 : 00 00 00 : 000000 (aux w  )   29 AN
50 : 00 00 00 : 000000 (aux w  )   30 AN  ; ;
52 : 00 00 00 : 000000 (aux w  )   30 AN
```

Fig. 6—Digital signal processor assembler "dsp.listing" file.

```
base = 16
size = 8
*Two 8iquad Sections in Cascade
0000: 00 08 d1 42
0002: 00 08 c0 37
0004: 00 08 80 01
0006: 00 08 9f ff
0008: 00 08 af fd
000a: 00 08 e0 01
000c: 01 08 20 00
000e: 00 08 40 00
0010: 04 41 00 10
0012: 04 81 00 19
0014: c4 48 dd 2f
0016: 04 48 47 2b
0018: 04 08 40 00
001a: 14 41 82 c1
001c: 94 88 40 00
001e: 04 48 c3 a6
0020: 04 48 6e e6
0022: 04 08 40 00
0024: 10 01 86 fa
0026: 90 08 40 00
0028: 00 09 00 01
002a: 00 01 00 40
002c: 00 08 00 0a
002e: 00 00 00 00
0030: 00 00 00 00
0032: 00 00 00 00
0034: 00 00 00 00
```

Fig. 7—File "b.out" written by DSP assembler.

Among the early improvements are additional macro-libraries and better syntax checking. The philosophy now in use is that of checking for specific errors. Because programmers are so ingenious at devising novel mistakes, it appears that the strategy should be reversed. It would be better to accept only code which conforms exactly to established forms, rejecting everything else.

More difficult, and correspondingly more valuable, are features that would simplify preparation of DSP programs for the user. This immediately suggests a compiler. However, the pipeline features of the DSP hardware will require the solution of design problems more complex than those for a standard compiler.

Register arithmetic is another area in which assistance to programmers would be valuable. Much of the speed advantage of the DSP comes from the planned use of automatically incremented registers for indirect addressing. Unplanned or random addressing of memory would forfeit this advantage. A compiler, then, should optimize the register use and incrementing. It might also have to change the locations in which the data are stored.

VII. EXAMPLES OF DSP ASSEMBLER INPUT AND OUTPUT

The following example was taken from a biquadratic filter program and shows the use of the macro library and preprocessor. The operations are primarily numerical calculations.

Figure 4 shows the input file required to program a two section filter.

```
Two Biquad Sections in Cascade
filetype i
0x0008
0xd142
0x0008
0xc037
0x0008
0x8001
0x0008
0x9fff
0x0008
0xaffd
0x0008
0xe001
0x0108
0x2000
0x0008
0x4000
0x0441
0x0010
0x0481
0x0019
0xc448
0xdd2f
0x0448
0x472b
0x0408
0x4000
0x1441
0x82c1
0x9488
0x4000
0x0448
0xc3a6
0x0448
0x6ee6
0x0408
0x4000
0x1001
0x86fa
0x9008
0x4000
0x0009
0x0001
0x0001
0x0040
0x0008
0x000a
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
0x0000
```

Fig. 8—File "d.out" written by DSP assembler.

The use of four macro calls reduces the amount of typing required of the programmer, and the probable number of errors. Figure 5 shows the messages the programmer receives on the terminal at the conclusion of a successful assembly. Figure 6 is the file "dsp.listing."

The files "b.out" and "d.out" are shown in Figs. 7 and 8, respectively. The file "d.out" is the input file to the simulator, DSPMATE, and the ROM programming utilities. The file "b.out" is a more readable output file, giving both ROM locations and machine language in hexadecimal machine language.

VIII. CONCLUSIONS

An assembler for the DSP differs from conventional assemblers in many interesting respects. Many of the standard principles of assembler design either do not apply or do not provide benefit. On the other hand, many novel problems arose, for which standard techniques were of little assistance.

The current version of the assembler is considered to be a useful, reliable tool for programmers to use today. There are several areas in which greater assistance for DSP programmers can be provided and improvements in those areas are anticipated.

REFERENCES

1. J. R. Boddie et al., "Digital Signal Processor: Architecture and Performance," B.S.T.J., this issue.
2. T. H. Crowley, "UNIX Time-Sharing System: Preface," B.S.T.J., *57*, No. 6, Part 2 (July–August 1978), pp. 1897–2304.
3. J. Aagesen, "Digital Signal Processor: Software Simulator," B.S.T.J., this issue.
4. M. E. Lesk, "Lex—A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories (October 1975).
5. S. C. Johnson and M. E. Lesk, "UNIX Time-Sharing System: Language Development Tools," B.S.T.J., *57*, No. 6 (July–August 1978), pp. 2155–75.

Appendix

Table I—Normal instructions

NOTHING	NOTHING	a = p	p = XSRC*YSRC
DEST = y	w = a	a = p + a	p = XSRC*w
DEST = YSRC		a = p − a	p = XSRC*c
DEST = w		a = p + 2*a	p = XSRC*abs (YSRC)
		a = p + 8*a	p = XSRC*abs (w)
		a = p + a/2	p = XSRC*c*sgn (YSRC)
		a = p + a/8	p = XSRC*c*sgn (w)
		a = p & a	

Notes:
(1) If YSRC occurs in column 4, DEST = YSRC may not be used in column 1. Instead, use DEST = y.
(2) If w is used in column 4, DEST = YSRC may not be used in column 1.
(3) If the second instruction following this one is a normal instruction in which XSRC refers to RAM, NOTHING must be selected for column 1.

Table II—Auxiliary arithmetic instructions

NOTHING	NOTHING	NOTHING	NOTHING
DEST = y	w = a	a = p	p = YSRC
DEST = YSRC	w = ltm1(w)	a = p + a	p = w
DEST = w	w = ltm2(w)	a = p − a	p = mtl1 (YSRC)
		a = p + 2*a	p = mtl2 ( )
		a = p + 8*a	
		a = p + a/2	
		a = p + a/8	
		a = p & a	
		a = a ≪ 14	
		a = a ≪ 18	

Note:
See Notes in Table I.

Table III—Nonarithmetic auxiliary instructions

NOTHING	NOTHING
DEST = y	REG = VALUE
DEST = YSRC	REG = &LABEL [*N*]
DEST = w	REG = &RAMVAR [*N*]
	REG = YSRC
	if (CONDITION) doset ( )
	if (CONDITION) doau ( )
	if (CONDITION) dowt ( )
	if (lc−−! = 0) doset ( )
	return

Notes:
(1) See Note 1 in Table I.
(2) An instruction containing only a semicolon is a no op.
(3) VALUE represents a number −− integer, real, octal or hex.
(4) &LABEL [*N*] represents the *N*th word in ROM memory after the address of the label. If *N* = 0, [*N*] may be omitted.
(5) &RAMVAR [*N*] represents the address of the (*N* + 1)th location in an array called RAMVAR, stored in RAM memory. If *N* = 0, [*N*] may be omitted.

Table IV—Conditions

ibf	IBUF full
obe	OBUF empty
c0	C0 = 1
c1	C1 = 1
a==0	a equal to zero
a < 0	a less than zero
a > 0	a greater than zero
1c! = 0	lc not equal to zero

Table V—Destinations (DEST)

Form 1	Form 2
obuf	out
*rda	rdz
*rda++	rdp
*rda−−	rdm
*rd++i	rdi
*rd++j	rdj
*rd++k	rdk

Table VI—Y sources (YSRC)

Form 1	Form 2
ibufy	iny
*rya	ryz
*rya++	ryp
*rya−−	rym
*ry++i	ryi
*ry++j	ryj
*ry++k	ryk

Table VII—X sources (XSRC)

Form 1	Form 2	
x	olx	previous value of *x*
VALUE	VALUE	immediate data
*rx++i	axi	RAM address
*rx++j	axj	RAM address
*rx++k	axk	RAM address
*rx	axz	RAM address
*rx++	axp	RAM address
*rx−−	axm	RAM address
ibufx	inx	
*(rom+rx++i)	rxi	ROM address
*(rom+rx++j)	rxj	ROM address
*(rom+rx++k)	rxk	ROM address
&LABEL [*N*]	&LABEL [*N*]	
&RAMVAR [*N*]	&RAMVAR [*N*]	

Note:
See Notes 3, 4, and 5 of Table III.

Table VIII—Registers (REG)

pc	program counter
rx	pointer for *x* data
ry	pointer for *y* data
rya	alternate pointer for y data
rd	pointer for write destination
rda	alternate pointer for write destination
i	auto-increment for memory pointer
j	auto-increment for memory pointer
k	auto-increment for memory pointer
lc	loop counter
auc	AU control
ioc	I/O control
syc	synchronization
str	status output

Note:
auc, ioc, syc, and str cannot be set by y sources.